In the United States Patent and Trademark Office

US Utility Patent Application for

Method and apparatus for adapting web contents to different display area dimensions

Inventor:	Vincent Wen-Jeng Lue 1484 Owen Sound Drive, Sunnyvale, California 94087 USA
File No.:	6154-01/LUE

Express Mail Label # EL 937345401 US Date of Deposit: **January 14, 2004**
I hereby certify that this paper or fee is being deposited with the United States Postal Service using "Express Mail Post Office To Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to "Mail Stop: New Application, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313"

Chi Ping CHANG
Name

[illegible]
Signature

Method and apparatus for adapting web contents to different display area dimensions

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is based upon and claims the benefit of U.S. Provisional Application No. 60/442,873, filed January 27, 2003.

BACKGROUND OF THE INVENTION

Field of the Invention

[0002] The present invention relates generally to automatic markup language based digital content transcoding and, more specifically, it relates to a method to simplify, split, scale and hyperlink HTML web content for providing a new method to repurpose legendary web content authored for desk top viewing to support smaller devices using limited network bandwidth such as palmtops, PDAs and data-enabled cell phones wirelessly connected with small display areas and processing capacities.

Description of the Related Art

[0003] With the popular use of Internet, vast and still growing amount of content have been made available through typical desktop browsers such as Internet Explorer (from Microsoft), Navigator (from AOL), and Opera (from Opera). They are coded in standard markup languages such as HTML and JavaScript. However, majority of them have been authored to fit regular desktop or notebook computers with large screen size, big processing capacity connected with high speed network.

[0004] As the web steadily increases its reach beyond the desktop to devices ranging from mobile phones, palmtops, PDAs and domestic appliances, problem in accessing legendary web content start to surface. Constraints from form factor and processing capacity render them practically useless on these devices. To solve this device dependency problem, one most cost effective approach is to provide intermediary adaptation in the content delivery chain.

[0005] Examples such as transcoding proxies can transform markup languages by removing HTML tags, reformatting table cells as text, converting image file formats, reducing image size, reducing image color depths, and translating HTML into other markup languages, e.g. WML, CHTML, and HDML. More involved approaches extract subsets of original content, either automatically or manually, or employ text summarizing techniques to condense the target content. Even more elaborated systems include client components using proprietary protocols between intermediaries and corresponding programs running in client devices to emulate standard browser interfaces, such as Zframeworks from Zframe Inc.

[0006] The main problem with conventional markup content transcoding is its inability to handle the sheer volume of content, both text and images, etc. inside the document for small devices. Arbitrary linear approach to partition the content based on markup language codes often makes the results unorganized with the original presentation intent lost. Summary techniques second guess the author's intent and are not able to always satisfy user's need.

[0007] Another problem with conventional markup content transcoding is its inability to handle common hidden semantics inside web documents such as HTML tables. However, authors are increasingly marking up content with presentation rather than semantic information and render the adapted content unusable.

[0008] Another problem with conventional markup content transcoding is its complexity in supporting new devices with different form factors. Instead of gracefully scaling the target transcoding result from small to large display devices, it relies on case-by-case settings requiring expensive development effort to support new devices.

[0009] Another problem with some conventional markup content transcoding is reliant on manual customizations to edit, select or annotate original content to assist adaptation process, which tends to be costly, error prone and not readily scalable.

[0010] Another problem with some conventional markup content transcoding is its dependency on specialized client software. Both deploying proprietary software to various client devices and administrating/configuring server adaptation engine increase cost significantly. This defies the original purpose of automatic content adaptation in place of adopting complete content re-authoring.

[0011] In these respects, Content Divide & Condense, the method to generate and scale document partitions with navigational links from single web content according to the present invention substantially departs from the conventional concepts and designs of the prior art, and in so doing provides an apparatus primarily developed for the purpose of providing a new method to transcode web content authored for desk top viewing into smaller ones to accommodate small display areas and capacities in mobile devices.

SUMMARY OF THE INVENTION

[0012] In view of the foregoing disadvantages inherent in the known types of markup content transcoding now present in the prior art, the present invention provides a new method, hereby named Content Divide & Condense, to simplify, partition, scale, and structure single content page onto hyperlinked and ordered set of content pages suitable for small device viewing before direct transcoding from HTML to the target markup language is applied, wherein the same can be utilized for providing a new method to transcode web content authored for desk top viewing into smaller ones to accommodate small display areas and capacities in mobile devices.

[0013] The general purpose of the present invention, which will be described subsequently in greater detail, is to provide a method to generate a minimum set of simplified and easily navigable web contents from a single web document, oversized for targeted small devices, while preserving all text, image, transactional as well as embedded presentation constraint information. Each of the simplified web content fits in display size and processing/networking capacity constraints of the target device. The whole set of generated pages are hyperlinked and ordered according to the intended two dimensional navigation semantics embedded inside the original content. A subset of XHTML is adopted to define the kind of content to be extracted from the original document. With the reduced content complexity in each partitioned page and the preserved navigational organization from original content, final set of documents after applying direct transcoding from each HTML partition to target markup language represent a much more accurate presentation with respect to the original content yet suitable for small device viewing.

[0014] To attain this, the present invention, named as Content Divide & Condense, generally comprises HTML parser, content tree builder, document tree builder, document simplifier, virtual layout engine, document partitioner, content scalar, and markup generator. The parser generates a list of markup and data tags out of HTML source document. It handles script-generated content on the fly and redirected content fetch similar to how common web browsers behave. Based on a specific set of layout tags, the builder constructs a content tree out of the markup and data tags. It interprets loosely composed HTML document following a set of heuristic rules to be compatible with how standard browsers work. This builder completes document tree build from the rest of markup and data tags on top of content element tree. It also adjusts the tree structure to be in compliant with XML specification without changing rendering semantics of the source HTML document interpreted by common browsers. The simplifier transforms the document tree onto an intermediate one defined by a subset of XHTML tags and attributes through filtering and mapping operations on tree nodes. Spatial layout constraints are heuristically estimated and calculated for data and image content embedded inside the document tree according to the semantics of HTML tags. Layout constraints include size, area, placement order, and column/row relationships. Based on the display size and rendering/network capacity constraints, the document tree is partitioned into a set of sub document trees with added hyperlinks and order according to the layout order and content structure. With target device display size constraint, each sub document tree is scaled individually by adjusting height and width attributes through the scalar. Source image references are modified if needed to assure server side image transcoding capability is leveraged. Each document tree defines a simplified HTML document which is generated during the markup generation step. Navigation order and hierarchical hyperlinks are assigned at the same time. The original content is thus represented by the set of smaller documents with hyperlinks and order defined between each other. Additional files such as catalog file indicating network bandwidth required for each document or text only document partitions can be generated and hyperlinked together in the same manner. Each simplified document can be transcoded onto target markup languages such as WML and cached by applying available direct transcoding technique.

[0015] There has thus been outlined, rather broadly, the more important features of the invention in order that the detailed description thereof may be better understood, and in order that the present contribution to the art may be better appreciated. There are additional features of the invention that will be described hereinafter.

[0016] In this respect, before explaining at least one embodiment of the invention in detail, it is to be understood that the invention is not limited in its application to the details of construction and to the arrangements of the components set forth in the following description or illustrated in the drawings. The invention is capable of other embodiments and of being practiced and carried out in various ways. Also, it is to be understood that the phraseology and terminology employed herein are for the purpose of the description and should not be regarded as limiting.

[0017] A primary object of the present invention is to provide a method to simplify, split, scale, and structure web content for small devices that will overcome the shortcomings of the prior art devices.

[0018] An object of the present invention is to provide a method to simplify web content to contain only the most primitive parts such as texts, images, forms, hyperlinks, and layout presentation arrangements etc., supported by standard markup language browsers for small devices.

[0019] An object of the present invention is to provide a method to extract web content to contain only the selected parts, such as text only with images as text links, or forms only, while preserving layout presentation arrangements etc. supported by standard markup language browsers for small devices.

[0020] Another object is to provide a method to split two dimensional layout arrangement such as tables, framesets and alignment to fit content display to the screen width constraint of the target device.

[0021] Another object is to provide a method to partition web content along both logical and embedded layout structure according to display area and capacity constraints of the target client device.

[0022] Another object is to provide a method to apply minimal scaling to each document partition individually to fit in target device display width constraint.

[0023] Another object is to provide a method to present the original web content by a set of hyperlinked and ordered document partitions according to the two-dimensional navigation order embedded inside the original document.

[0024] Another object is to provide a method to utilize target device display size and resource capacities to partition the document by conducting virtual layout against the original content represented by a markup language.

[0025] Another object is to provide a method to present a hyperlinked catalog content indicating the required network bandwidth required for accessing each document partition from the target device.

[0026] Other objects and advantages of the present invention will become obvious to the reader and it is intended that these objects and advantages be within the scope of the present invention.

[0027] To the accomplishment of the above and related objects, this invention may be embodied in the form illustrated in the accompanying drawings, attention being called to the fact, however, that the drawings are illustrative only, and that changes may be made in the specific construction illustrated.

BRIEF DESCRIPTION OF THE DRAWINGS

[0028] Various other objects, features and attendant advantages of the present invention will become fully appreciated as the same becomes better understood when considered in conjunction with the accompanying drawings, in which like reference characters designate the same or similar parts throughout the several views, and wherein:

[0029] **Figure 1** is to illustrate the function of Content Divide & Condense;

[0030] **Figure 2** is to show Content Divide & Condense working as part of a transcoding server;

[0031] **Figure** 3 is to show Content Divide & Condense working as part of a proxy server;

[0032] **Figure 4** is to show Content Divide & Condense working as part of a web server;

[0033] **Figure 5** is to show affects and steps of content reduction engine;

[0034] **Figure 6** is an example showing sample HTML code and the corresponding markup and data list;

[0035] **Figure 7** is to show steps to parse source HTML document into markup and data element list;

[0036] **Figure 8** is to show steps to build layout element tree from markup and data element list;

[0037] **Figure 9a** and **Figure 9b** are a set of handlers for inserting implicit tags;

[0038] **Figure 10** is additional set of handlers for inserting implicit tags;

[0039] **Figure 11** is to show three types of relationships between tag node and associated markup or data element;

[0040] **Figure 12** is an example of HTML source code and its corresponding content element tree;

[0041] **Figure 13** is to show steps to build a document tree from the corresponding content element tree;

[0042] **Figure 14** is to show steps to build document sub-tree from markup and data element list between content nodes;

[0043] **Figure 15** is to show steps to rectify an HTML document tree to an XML compliant one;

[0044] **Figure 16** is to show steps to handle non-xml-compliant style tags;

[0045] **Figure 17a** and **Figure 17b** are to demonstrate node insertion operations;

[0046] **Figure 18** is a sample of HTML code, its corresponding preliminary document tree and XML compliant document tree;

[0047] **Figure 19** is an example to show handling of non-XML compliant style tags;

[0048] **Figure 20** is to show steps to handle non- XML compliant form tags;

[0049] **Figure 21** is an example of source HTML code and the corresponding document tree after form and style tag handling;

[0050] **Figure 22a** and **Figure 22b** show steps to map document tree onto a simplified one based on a subset of XHTML tags;

[0051] **Figure 23a** and **Figure 23b** show continuous steps to map document tree onto a simplified one based on a subset of XHTML tags according to **Figure 22a** and **Figure 22b**;

[0052] **Figure 24** is an example to demonstrate simplification of map tags onto list tags;

[0053] **Figure 25** is a frameset HTML code sample and its corresponding document tree;

[0054] **Figure 26** is a sample of document tree and its corresponding HTML code from frameset simplification;

[0055] **Figure 27** shows steps to apply layout and style constraints;

[0056] **Figure 28** shows virtual layout steps to assign sizing information to document node;

[0057] **Figure 29** shows steps to calculate layout sizing parameter through placement constraint;

[0058] **Figure 30** shows steps to split and partition document tree based on estimated layout width and content size;

[0059] **Figure 31** shows steps to split a document node with oversized layout width;

[0060] **Figure 32** shows steps to partition document against an oversized document node on sizing parameters A and N;

[0061] **Figure 33** shows steps to split a node and an example;

[0062] **Figure 34** shows steps to create document partition based on a set of descendant content nodes from placement constraint;

[0063] **Figure 35** is an example to show document tree change before and after node split without document partitioning;

[0064] **Figure 36** is a document partition example;

[0065] **Figure 37** is an example of document data node partition;

[0066] **Figure 38** shows steps to scale document partition;

[0067] **Figure 39** shows steps to calculate minimum width required for a column of document node;

[0068] **Figure 40** shows steps to obtain navigation order of a document tree; and

[0069] **Figure 41** shows a sample of hyperlinks and navigation order among document partitions.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0070] Turning now descriptively to the drawings, the attached figures illustrate a method to generate and scale document partitions with navigation links from single web content for small device viewing, as shown in **Figure 1** which comprises HTML parser **16**, content tree builder **18**, document tree builder **20**, document simplifier **22**, virtual layout engine **24**, document partitioner **26**, content scalar **28**, and markup generator **30**. The parser **16** generates a list of markup and data tags out of HTML source document **12**. It handles script generated content on the fly and redirected content fetch similar to how common web browsers behave. Based on a specific set of layout tags, the builder **18** constructs a content tree out of the markup and data tags. It interprets loosely composed HTML document following a set of heuristic rules **34** to be compatible with the manner how standard browsers work. This builder **18** completes document tree build from the rest of markup and data tags on top of content element tree. It also adjusts the tree structure to be in compliant with XML specification without changing rendering semantics of the source HTML **12** document interpreted by common browsers. The simplifier **22** transforms the document tree onto an intermediate one defined by a subset of XHTML tags and attributes through filtering and mapping operations on tree nodes. Spatial layout constraints are heuristically estimated and calculated for data and image content embedded inside the document tree according to the semantics of HTML tags. Layout constraints include size, area, placement order, and column/row relationships. Based on the display size and rendering/network capacity constraints, the document tree is partitioned into a set of sub document trees with added hyperlinks and order according to the layout order and content structure. With target device display size constraint **32**, each sub document tree is scaled individually by adjusting height and width attributes through the scalar **28**. Source image references are modified if needed to assure that the server side image transcoding capability is leveraged. Each document tree defines a simplified HTML document which is generated during the markup generation **30** step. Navigation order and hierarchical hyperlinks are assigned at the same time. The original content is thus represented by the set of smaller documents with hyperlinks and order defined between each other. Additional files such as catalog files indicating network bandwidth required for each document or text only document partitions can be generated and hyperlinked together in the same manner. Each simplified document can be transcoded onto target markup languages such as WML and cached by applying an available direct transcoding technique.

[0071] Turning now to **Figure 2**, overall effects of Content Divide & Condense **36** is illustrated. Input to the engine is a single web document such as HTML page **38**. The engine then generates a set of simplified and small HTML documents **40** hyperlinked together. A linear navigation order is also assigned to each partition document.

[0072] The engine could process the same document with more than one settings at the same time. For example, it generates the partition both with and without images to allow the flexibility to turn on or off the image content while ensuring device capacity is fully utilized. The same text paragraph could appear in two partitions, one consists of only text data and the other contains also image links. Because image capacity is replaced by text data, these two partition documents can not be transcoded directly between each other by adding or removing image links. However, cross links can be inserted such that it is possible to access text data as preview and retrieve full image embedded one when interested.

[0073] Systems with Content Divide & Condense working together with client device and other servers are shown in **Figure** 3, **Figure** 4, and **Figure** 5, as examples. Referring to **Figure** 3, a transcoding server consists of HTTP handler component **46**, a local cache **48**, HTML transcoder **50** and Content Divide & Condense **42**. The client **44** sends an HTTP request **52** along with client agent id to the transcoding server with an URL referencing a web content from the Web Server **56**. The HTTP Handler **46** then sends another HTTP request **54** to the Web Server **56** for the document identified by the URL from the client request **52**. Web Server **56** returns the document to the HTTP Handler **46**, which passes the document along with a client agent id through Content Divide & Condense **42**, which generates a set of hyperlinked and simplified document partitions along with pre-fetched and properly scaled images. Each partition is then passed through an HTML Transcoder **50** to map HTML onto target ML language for the client and stored in the local cache **48** along with scaled images. The HTTP Handler **46** selects the first page from the local cache **48** and returns to the client **44** as part of HTTP response **58**.

[0074] Referring to **Figure 4**, Content Divide & Condense **60** works as part of an HTTP Proxy Server. It works similarly as in **Figure 3**. The differences are source link updates and HTTP request/response cache. There is no need to resolve absolute source links when working as the HTTP Proxy Server. By default, HTTP request/response pairs are cached and cache hit is always checked before making remote fetch.

[0075] Referring to **Figure 5**, Content Divide & Condense **62** works as part of an HTTP Server. The client **64** sends an HTTP request **66** along with the client agent id to the server. The HTTP Handler **68** fetches the target HTML document **70** from local storage. Based on the client agent id, the HTTP Handler **68** determines whether to send the document back directly or pass the client agent id and document to Content Divide & Condense **62**. If transcoding is needed, Content Divide & Condense **62** generates a set of hyperlinked and simplified document partitions along with properly scaled images. Each partition is then passed through an HTML Transcoder **72** to map HTML onto target ML language for the client **64** and stored in the local cache **74** along with scaled images. HTTP Handler **68** selects the first transcoded page from the local cache **74** and returns to the client **64** as part of HTTP response **76**.

[0076] The HTML parser translates input HTML document into a list of markup and tags similar to what common browsers do. Each element of the list is either a markup with its attributes or a block of raw data, such as text data or script codes. An example of HTML code sample **77a** and its corresponding markup and data list **77b** is shown in **Figure 6**. The overall steps are shown in **Figure 7**, including a syntactic parser **78**, frame source handler **80**, script source handler **82** and final tag list selector **84**.

[0077] When the parser **78** encounters <FRAME> tag, source links inside the tag are resolved and corresponding document fetched **80a**/parsed **78** on the fly. <FRAME> source is inserted into the original tag list right after the corresponding <FRAME> tag with an added </FRAME> tag at the end to enclose it. The process continues recursively as shown in **Figure 7**.

[0078] When the parser **78** encounters <SCRIPT> tag, JavaScript source codes are executed by a JavaScript engine **86** with a simplified document object model **88**. Source links are followed to fetch remote codes **82a**, if there is any. The simplified document object model **88** supports both document.write and document.writeln functions and is capable of generating HTML content **86a** on the fly. The in-line generated codes, if there is any, are parsed by the parser **78** and the resulting tag list is inserted right after the corresponding <SCRIPT> tag. This process runs recursively as shown in **Figure 7**. The document object model could be expanded when needed by implementing additional objects and functions, including handling of specific client and/or user browser settings, cookies, etc.

[0079] After parser **78** exhausts all input sources, HTML tags requiring exclusive-or selection or filtering are handled before final list **90** is generated. They include <SCRIPT> vs. <NOSCRIPT>, <FRAME> vs. <NOFRAME>, and <EMBED> vs. <NOEMBED>, <LAYER> vs. <NOLAYER>. The parser tag selection **84** ignores <NOSCRIPT>, <NOFRAME>, and <EMBED> tags. These tags and all source markup data enclosed are left out from final tag list. Capability for the parser tag selection **84** to select an intended subset of tag list from the source document could readily be added. Depending on target client device context and document semantics, the parser might have an option to choose <NOFRAME> instead of <FRAME>, <NOSCRIPT> instead of <SCRIPT>, <EMBED> instead of <NOEMBED>, <LAYER> instead of <NOLAYER>, etc. Additional tags accepted as standards moving forward could also be supported in the similar manner.

[0080] The content tree builder constructs a tree out of the set of content markup elements based on the tag list generated by the parser. An HTML tag is considered content element if it designates directly an actual layout area when the content is rendered. The set of HTML tags considered content elements are listed in Table 1(a). These tags are different from those specifying mainly display styles, user interface context, or executable script codes such as those shown in Table 2(b). The set of content tags are focused first to simplify handling of many loosely composed HTML documents where style and context tags are not required to follow strict XML structures.

[0081] Table 1(a) A Table of Content Tag List for HTML

Content Tag							
a	abbr	acronym	address	applet	area	base	blockquote
body	br	button	caption	col	colgroup	del	dfn
dd	dir	div	dl	dt	embed	fieldset	frame
frameset	h1 through h6	head	html	hr	iframe	ilayer	img
input	ins	isindex	label	layer	legend	li	link
map	marquee	menu	meta	multicol	noembed	noframes	noscript
object	ol	optgroup	option	p	param	pre	samp
select	spacer	span	style	table	tbody	td	textarea
tfoot	th	thead	title	tr	ul	xmp	wbr

[0082] Table 1(b) A Table of Style/Context Tag List for HTML

Style and Context Tag											
b	basefont	bdo	big	blink	center	cite	code	em	font	form	i
kbd	q	s	small	strike	strong	sub	sup	tt	u	var	

[0083] The steps to build content tree **104** is shown in **Figure 8**. Each element of markup and data list **92** generated by the parser is visited in order **92a**. Those not belonging to content tags, including data elements, are ignored **94**. Conditions are then checked for the need to insert implicit tag **96** into the list to ensure consistency between layout semantics and document tree structure. After new tag is inserted **96a**, the process returns back to the list **92a** and repeats from the new one on. If the current tag is a start tag **98**, it is pushed to the top of stack **98a** while an end tag requires additional handling. Normally, the top of stack would match an end tag encountered. Otherwise **100**, the whole stack is examined to see if there is matching one **101**. If yes **102**, the stack is popped **102a** until the matching one is reached. An end tag without any matching tag in the stack is ignored.

[0084] Implicit tags are generated on the fly as shown in **Figure 9a**, **Figure 9b** and **Figure 10**. A state refers to the name of the tag at the top of the stack. All the rules from (a) **106** to (n) **108** specify conditions when implicit end tags are detected. Rules (a) **106**, (e) **110**, (f) **112**, and (g) **114** describe how <TR> tag is implied as well. Rule (k) **116** is needed because the parser fetches frame source and inserted the tag list after the associated <FRAME> tag followed by an added </FRMAE> one. The last rule (n) **108** states that if </BODY> tag is encountered **108a** without a matching state **108b**, the end tag of the current state, if needed, is added automatically **108c**. The list of HTML tags without end tags are <AREA>, <BASE>, <BR>, <COL>, <COLGROUP>, <FRAME>, <HR>, <IMG>, <INPUT>, <ISINDEX>, <LINK>, <PARAM>, and <BASEFONT>. These rules essentially implement what specified by HTML standard.

[0085] The document tree is built during popping tags from the stack. A tree node is defined after the top element is popped from the stack. There are three possible kinds of nodes, as shown in **Figure 11**, depending on the relationships between a tag node and its associated markup elements. The most common one is formed by a paired start **120** and end **122** elements as in (a) **118**. A degenerated one could be like (b) **124** where a node is formed by a single element **126** or (c) **128** where a data node represents the data element **130** in between two adjacent markup elements from the input list. An example of sample HTML code **131a** and its corresponding content tree **131b** is shown in **Figure 12**.

[0086] The set of tags considered content elements and the set of rules for determining existence of implicit tags are expected to be updated and evolve. As this design is to support legendary web content, it needs to be as lenient to document not following exactly HTML specs as common browsers are. Evolution of browser markup languages would also force new updates, hence new changes in rules and setting as discussed here.

[0087] Based on content element tree **132**, the remaining non-content markup and data tags are handled to complete the document tree **134**. Firstly, these tags are visited following the steps shown in **Figure 13** together with **Figure 14** to complete a preliminary document tree **136**. Then, a set of adjustments are applied to special set of non content tags according to the underlying HTML semantics to the final document tree **138** to be compliant with XML structure as shown in **Figure 15**.

[0088] Based on the content element tree **132**, each node is visited following a depth first order **132a** and sub document trees, based on non-content tags, are built **142** and inserted **144** onto the content element tree **132** to form a preliminary document tree **134**. The steps shown in **Figure 13** build sub document trees **142** based on segments of tag lists partitioned by content tags in the content element tree **132**. A list of non-content tags is defined between the first child node and parent node, two neighboring sibling nodes, the last child node and parent node, or simply a single terminal node. A set of sub document trees **142** are constructed out of each such segment of tags and inserted as new child nodes **144** of the defining parent node in order.

[0089] The tree building steps are shown in **Figure 14**, similar to **Figures 9a** and **9b** but a bit simplified. The main difference lies in the handling of end tag without matching start tag in the stack. A tree node with this single end tag is created instead of being removed. This happens often because HTML does not require strict XML structure on style and context tags and pairing start/end tags might belong to two different segments of tag lists partitioned by content element tree nodes. This process on each segment of tag list results in an ordered list of sub trees **146** to be inserted back to the content element tree **132**. A sample HTML code **147**, its corresponding preliminary document tree **147a** and XML compliant document tree **147b** are shown in **Figure 18**.

[0090] Steps to rectify preliminary document tree **136** to be XML compliant are shown in **Figure 15**. It iterates through the list of leaf nodes (node without any children) in order **148** and calls proper handlers for different types of nodes. Three handlers are considered here. If the leaf node is associated with an end tag **150**, it is regarded as extra end tag and removed from the document tree **150a**. If the leaf node is associated with a form tag **152**, a form tag handler **152a** is called. Otherwise, if it is a text style tag **154**, a style tag handler **154a** is called.

[0091] There are four tree insertion operations employed in the handler as shown in **Figure 17a** and **Figure 17b**. The insertion adds a new child node to a parent node but reset a set of original child nodes from the parent node to itself. Assuming a new node F, and a parent node P, **Figure 17a** (a) shows the operation of inserting F as the right of B under P **156**, where P is an ancestor node of B. **Figure 17a** (b) shows the operation of inserting F as the left of D under P **158**, where P is an ancestor node of D. **Figure 17b** (c) shows the operation of inserting F as between A and E under P **160**, where P is ancestor node of both A and E. And **Figure 17b** (d) shows the operation of inserting F between B and D under P **162**, where F becomes ancestor node of both B and D as a child of P.

[0092] Detailed steps of text style tag handler are shown in **Figure 16**. It follows the rule that style specification does not pass <TD>, <BODY>, or <HTML>. In this implementation, multiple <BODY> or <HTML> nodes are possible because of pre fetched frames. An attempt is made to locate the matching leaf node among the rest of leaf nodes following this range rule first **164**. If none is found, the style effect is assumed to cover the rest of document element under the first ancestor <TD>, <BODY>, or <HTML> node **166**. Existence of leaf node of the same non-ending style tag is also considered an implicit matching end tag. Once matching node pairs are identified, the closest common parent node is located **166a** and the new style nodes are inserted accordingly to cover all elements enclosed by these two nodes under this common ancestor node.

[0093] The form handler follows the steps shown in **Figure 19**. When a leaf FORM tag node is encountered **168**, it tries to search for all form element nodes belonging to this form. Form element nodes are those with tags such as <INPUT>, <SELECT>, <FIELDSET>, <OPTION>, <OPTGROUOP>, and <TEXTAREA> which are expected to be enclosed by a matching pair of <FORM> tags in the source document. These nodes are content elements and have already been built into the document tree. Depth first order search following the leaf <FORM> node is required to collect them. The search ends either normally or with an error. If another <FORM> node is encountered **170** before a matching end <FORM> node is found, it is considered document error. Otherwise, the search continues and collects a list of <FORM> element nodes **172** until the matching end <FORM> leaf node is found **174**. If the search exhausts the tree, it is assumed an implicit end <FORM> tag is intended right before </BODY> tag. When the search ends without error, it checks to see if the list of <FORM> element nodes is empty **174a**. If so, no <FORM> node is needed **176**. Otherwise, a new <FORM> node is created based on matching pair of <FORM> tags of the leaf <FORM> node **178**. The least common ancestor node **180**, which is neither <TABLE> nor <TR> nodes, among the collected list of <FORM> element nodes is then found. The new <FORM> node is then inserted between the first and last nodes of the collected node list under this least common ancestor found **182** as depicted in **Figure 17b** (d) **162**.

[0094] A sample document tree **183** built by the above stated steps from sample content element tree **131b** in **Figure 12** is shown in **Figure 20**. The handlers are provided for correcting loosely structured HTML document. Heuristic assumptions are made in these handlers with respect to when erroneous documents are encountered. More handlers could be added for other conditions not discussed above. In addition, assumptions on how browsers behave might also evolve as new versions and/or new kinds of browsers continue to be adopted in the market.

[0095] The simplifier transforms the document tree onto an intermediate one defined by a subset of XHTML tags and attributes through filtering and mapping operations on tree node. A document tree is condensed and simplified based on a subset of XHTML 1.0 markup tag list specified in Table 2. The main objective of this design is to render the content in terms of document tree while preserving as much as possible the intended content, style, hyperlinks and form interactions. Markup tag associated with original document tree node could belong to HTML, XHTML, or even generic XML. The simplification process goes through each node and performs transformation or filtering against a node or a sub tree. Semantics of HTML and XHTML tags are embodied in these transformation rules.

[0096] Table 2 A Table of Simplified HTML Tags

Name	Attributes	Description

A	href = URL
	name = CDATA
	rel= Link Type
	rev = Link Type
	type = Content Type
		anchor

ABBR		abbreviation (e.g. WDVL)
ACRONYM		
ADDRESS		information on author
B		bold text style

BASEFONT	size = CDATA
	color = Color Number
	face = CDATA
		base font size

BIG		large text style
BLOCKQUOTE	CITE = URL	long quotation

BODY	alink = Color Number
	background = URL
	bgcolor = Color Number
	link = Color Number
	text = Color Number
	vlink = Color Number
		document body

BR		forced line break

BUTTON	disabled
	name = CDATA
	type = button | submit | reset
	value = CDATA
		push button

CAPTION	align = calign	table caption
CENTER		shorthand for DIV align = center

CITE		citation
CODE		computer code fragment
DD		definition description
DFN		instance definition
DIR	compact	directory list
DIV	align = left | center | right | justify	generic language/style container
DL	compact	definition list
DT		definition term
EM		emphasis
FIELDSET		form control group

FONT	color = Color Number
	face = CDATA
	size = CDATA
		local change to font

FORM	action = URL
	accept-charset = Charset
	enctype = Content Type
	method = get | post
		interactive form

H1,H2,H3,H4,H5,H6	align = left | center | right | justify	heading
HEAD	profile = URL	document head, contains BASE, LINK, META, SCRIPT, STYLE, TITLE.
HR	noshade	Horizontal rule

HTML	version = CDATA
	lang=Language Code
		document root element

I		italic text style

IMG	alt = Text
	src = URL
	height = Length
	longdesc = URL
	width = Length
	align = top | bottom | middle | left | right
		Embedded image

INPUT	accept = ContentText
	alt = CDATA
	checked
	disabled
	maxlength = Number
	name = CDATA
	readonly
	size = CDATA
	type = Input Type
	value = CDATA
		form control

KDB		text to be entered by the user
LABEL		form field label text
LEGEND	align = lalign	fieldset legend

LI	type = li style
	value = number
		list item

MENU	compact	menu list

META	content=CDATA
	http-equiv = Name
	scheme = CDATA
		generic meta information

OBJECT		generic embedded object

OL	compact
	start = Number
	type = ol Type
		ordered list

OPTGROUP	label = Text disabled	option group

OPTION	diabled
	label = Text
	selected
	value = CDATA
		Selectable choice

P		Paragraph
PRE		preformatted text
Q	cite = URL	short inline quotation
S		strike-through text style
SAMP		sample program output, scripts, etc.

SELECT	disabled
	multiple
	name = CDATA
	size = Number
		option selector

SMALL		small text style
SPACER		generic language/style container
SPAN		generic language/style container
STRIKE		strike-through text
STRONG		strong emphasis
SUB		subscript
SUP		superscript

TABLE	bgcolor = Color Number
	border= Pixels	

	frame= Tframe
	summary= Text	

TD	abbr = Text
	sxis = CDATA
	bgcolor = Color Number
	colspan = Number
	rowspan= Number
		table data cell

TEXTAREA	cols = Number
	rows = Number
	disabled
	name = CDATA
	readonly
		multi-line text field

TH	abbr = Text
	axis = CDATA
	bgcolor = Color Number
	colspan = Number
	rowspan = Number
		table header cell

TITLE		Document title
TR		table row
TT		teletype or monospaced text style
U		underlined text style
UL	compact type = Ul Style	Unordered list
VAR		instance of a variable or program argument

[0097] The simplification steps are shown in **Figure 21**. It walks through the document tree following depth first order starting from the root node **184**. If it is a data node **186**, a simple filtering process is applied to remove consecutive space, carriage returns or line feeds **186a**. Otherwise **188**, the rooted sub tree is removed **188a** if the associated tag belongs to the set of five types, <APPLET> **190**, <SCRIPT> **192**, <NOSCRIPT> **194**, <NOFRAME> **196**, or <NOLAYER> **198**. They are ignored because of 1. Java support as activating specific client application is not covered; 2. Script codes have either been executed to get dynamic client content or not yet supported; 3. <NOSCRIPT>, <NOFRAME>, or <NOLAYER> do not usually contain useful information, but simply advisory or warning messages.

[0098] For <META> tags **200**, only those with the presence of HTTP-EQUIV attribute are retained **202**. Other Meta tags used for naming, keywords or other purposes are removed **204**, as they do not have significance either on content or how content is fetched. Response information is extracted from the HTTP attribute value pair denoted by the values of HTTP-EQUIV and CONTENT attributes, and stored as part of document context information **206**, such as document encoding and language set specification.

[0099] Table simplifier **208** is applied for <TABLE> nodes as described in **Figure 22b** (b) **209**. It goes through its direct child nodes and ensures that: 1. <THEAD> is placed before <TR>, <TBODY>, and <TFOOT>; 2. <TFOOT> node is placed after <THEAD>, <TR>, and <TBODY> nodes; 3. order of <TR> nodes are kept the same, if there are <THEAD>, <TBODY>, or <TFOOT> nodes.

[0100] A node belonging to four types of tags are replaced by <DIV> node **210** to keep the structure in place while retaining the enclosed data. <ILAYER> **212** and <LAYER> **214** are used for positioning a block of content. This will not be proper after splitting and scaling the content. <MARQUEE> **216** is used for animating a block of content, not supported by most browsers. <OBJECT> **218** is to activate embedded client application and not handled by the simplification process. Alternate text enclosed by the <OBJECT> tags is preserved. The simplification process ignores presentation and functional controls intended by these tags and keeps only the content data as a division block.

[0101] When <BASE> node is encountered **220**, the document context is updated **222** on the originating source URL. This node is removed afterwards **224**, as the resulting content would be sent from servers of different URL.

[0102] An <INPUT> node with type attribute value FILE or IMAGE is removed **226**. Image based input button might require client side image mapping capability which would be distorted during scaling.

[0103] <FRAMESET> node is handled by Frameset simplifier **228** as shown in **Figure 22a** (a) **230**. Contents enclosed by frameset and frame tags intended for separate client display windows are replaced by table structure preserving similar layout constraint. In essence, it removes interactions between frame windows on the client device while ensuring the original frame set content is properly displayed. A table node is created **234** for a <FRAMESET> node **232**. Depending on ROWS attribute specification **236**, one or multiple <TR> nodes are added to this table node. Single row table is assumed without the presence of ROWS attribute. Similarly, depending on COLS attribute specification **238**, one or multiple <TD> nodes are added to each <TR> node. Again, one column row is assumed without COLS specification. Frame source content as children nodes of original <FRAME> node are reattached to the corresponding <TD> node as their parent node **240**. However, when the child of <FRAMESET> node is also another <FRAMESET> node **242**, it is reattached to the corresponding <TD> node as its child node. The resulting <TABLE> node rooted tree is connected to the document tree in place of the <FRAMESET> node **244**. An example of mapping from <FRAMESET> node tree to <TABLE> node tree is demonstrated in **Figure 25** and **Figure 26**.

[0104] <TR> node is handled by TR simplifier as shown in **Figure 22b** (c) **246**. If there is background color specified for the whole row, this attribute BGCOLOR is duplicated to each <TD> node **248** under this <TR> node, when no BGCOLOR is specified for the <TD> node. As table structure could be split and changed, this attribute will be honored at <TD> node but not <TR> node.

[0105] <MAP> node is handled by the map simplifier as shown in **Figure 23a** (d) **250**. The navigation links embedded inside a map are replaced by a newly created list of hyperlinks. In essence, a <MAP> node **251** is replaced by a <UL> node **252** and each <AREA> node under <MAP> node is replaced by a list <LI> node **254** with an anchor <A> node **256** presenting the reference specified in HREF attribute **258** inside <AREA> tag. Hyperlinked text for each <AREA> node is determined by its ATL attribute **260**, if present. Otherwise, the file name of the URL specified by HREF attribute is used **262** instead. The resulting <UL> rooted document tree is then stored in the context **264** indexed by the name of <MAP> node through NAME attribute. <MAP> node rooted tree is then removed from the document **266**. This is demonstrated in **Figure 24** with an example <MAP> rooted tree **266a** and its corresponding <UL> rooted tree

266b.

[0106] <IMG> node **267** is handled by the img simplifier as shown in **Figure 23b** (e) **268**. If it is a server side image map, as specified by ISMAP attribute, this node is removed **270**. Because of possible scaling, image map is not supported. If the node is a client side image map, this node is indexed in the content **272** for possible replacement later with corresponding <MAP> tree.

[0107] <IFRAME> node **273** is handled by IFrame simplifier as shown in **Figure 23b** (f) **274**. A newly created single cell table replaces the original <IFRAME> tag **276**. To distinguish alternate text enclosed by <IFRAME> tags, the fetched frame source content is enclosed by the parser with <DIV> tags. The figure describes detailed steps on how the table tree and the frame source content are connected and inserted into the document tree while <IFRAME> and alternate text is removed.

[0108] If a node does not match any of tags considered above, it is checked against the list in Table 2. Those with tag names not preset in this table are removed from the document tree **278**. Then its attributes are updated **280** as shown in **Figure 21**. Those attributes not listed in Table 2 are removed. All relative URL as in HREF or ACTION attributes are resolved according to document context with its absolute path. Actual font size has to be used for SIZE attribute associated with FONT node as well. Because of the presence of <BASEFONT>, relative font size can be resolved with the help of document context.

[0109] After walking through the whole document tree nodes, each <IMG> node with USEMAP attribute indexed by a map name, is further condensed **282** as shown in **Figure 21**. The <IMG> node rooted tree in the document is replaced by the corresponding <UL> rooted tree created from original <MAP> node, if there is any, or removed if there is none. This is the last step to complete the simplification process.

[0110] Changes in the target tag and attribute list as well as how different types of document nodes are handled would result in variations of document tree reduction. For example, the data filter could employ a scheme to retain only content for hyperlinks or form interface but removing all others. Another example is the support of <STYLE> tags for getting more precise information and better control on how document would be rendered at client devices. Yet another example is support for international language attributes inside markup tags in addition to those from HTTP headers. As standards of markup language evolve, changes are expected to accommodate new developments.

[0111] Spatial layout constraints are heuristically estimated and calculated for test and image content embedded inside the document tree according to the semantics of HTML tags. Layout constraints include size, area, placement order, and column/row relationships. Display size and client capacity requirements are estimated for the simplified document through virtual layout on the underlying document tree. These parameters are used to determine how the document should be partitioned and scaled to accommodate a target client device. The process of virtual layout includes assigning placement constraints and calculating layout sizing information for each content node based on the constraints and a set of layout parameter settings.

[0112] Given a document tree, virtual layout determines the set of content children for each content node and assigns it placement constraint among these children nodes. A set of nodes C1, C2,.. Cn form content children set S of a node N if 1. N is ancestor node of each node Ci in S and 2. each node Ci in S is either content node or data node and 3. for all leaf nodes under N rooted tree, there exists one and only one node in S as its ancestor node. By default, the content children set of a content node is defined as the collection of highest-level offspring content/data nodes. Virtual layout assigns placement constraint to document tree nodes such that 1. every leaf node of the document tree belongs to one and only one content children set and 2. each content node belongs to at most one content children set.

[0113] To estimate the minimum display width needed for content rendering, placement constraint is designated to content nodes. Placement constraints adapted here are either table with rows/columns or simply a single column. Steps to assign placement constraint are illustrated in **Figure 27**. It visits each node following depth first order **284** and adds row/column constraint to <TABLE> node on the associated <TD> node **286** accordingly, including row and column span. <TR> node is ignored **288** as its layout semantic has been considered when assigning row/column constraint for its parent <TABLE> node. All other content nodes, except for leaf ones **289**, are assigned single column placement constraint **290** on its content children set.

[0114] Four sizing parameters could be derived from the document tree with placement constraints assigned and display font sizes selected for the target client device. They are scalable width (W) in pixel, minimum width (M) in pixel, image area (A) in square pixel, and total number of characters (N). W represents size required for scalable layout components such as <IMG> and <TEXTAREA>, for example. M characterizes the minimum fixed layout component needed. It is typically the width of the longest word in the document text. A is the total area of all images in the document. N is the number of all display characters inside the document, symbolizing the amount of text information carried. The minimum display width D required for rendering a document rooted at a node with W and M will be W+M.

[0115] Font size and language settings are needed to calculate layout sizing information. Character and word boundaries are determined by language encoding for the content text data. Average width of character is dependent on the specified font family and font size. To simplify the layout process, a single font family with minimum and default font size is indexed by the client agent and language code. For example, English content from IPAQ IE browser would use Times Roman font with minimum font size 2 and default font size 3. Selection of these parameters is to be as realistic as possible and depends on the settings of specific user agent.

[0116] A layout context is referenced and updated when visiting each node. Included in this context are current font size, layout sizing constraint (Nmax, MWmax, Amax), NoFlow flag, and Atomic flag, etc. Nmax is the maximum value of N allowed for the whole document. MWmax is the maximum (W+M) value for the whole document. Amax is the maximum image area allowed, NoFlow flag is used when text characters would be laid out in one line. And Atomic flag means no partition is allowed. Style nodes such as <FONT> node affect the font size. <FORM> node enables Atomic flag, meaning elements of <FORM> tags should belong to the same document. <SELECT> node enables NoFlow flag to indicate text in a data node, mainly under <OPTION> node, should be shown in one single line.

[0117] Steps to calculate sizing parameter values for a document node associated with placement constraint are shown in **Figure 28**. Initial values of W, A, N, and M are set to zero for all nodes. A bottom up process is employed to propagate layout sizing information from leaf nodes to the root through these constraints. Starting from the root, it walks down the tree in a depth first order to size each node. For non-leaf node **292**, it is checked if layout context, including font size, and character flow control, needs to be updated. A leaf node **294**, which is either data node, containing only character text, or image node, linking an image source, provides the basic layout dimension data. An <IMG> node **296** with width w and height h would have size W = w, A = w*h, N = 0, and M = 0. A data node **298** without NoFlow flag on text layout context total number of characters t and the longest word in terms of characters l would have size N = t, M = l*F, W = 0, and A = 0, where F is the average character width under the current font setting in the text layout context. If NoFlow flag is set for text layout context, as in the case under <SELECT> node rooted tree, M is assigned as t*F instead of l*F. More precise calculation is possible when equipped with detailed display size information for each character instead of using average with.

[0118] After all children of a content node have been sized, the associated placement constraint is applied to obtain sizing information **300** for this node. Generic steps to calculate these parameter values according to the constraint are shown in **Figure 29**. A slight variation is applied for <SELECT> node. (W, A, N, M) is initially set to (0,0,0,0) **302** during the calculation. Each row is iterated through **304** to update these four values. The number of nodes in a row could be smaller than the number of columns in the constraint because of column span consideration. Area A and total number of characters N are additive but considered only once when spanning multiple rows. M and W are assigned such that both (M+W) and M should both be maximum among all rows.

[0119] Propagation function could be node specific. For <SELECT> node, the minimum of all M values among all its <OPTION> child nodes is assigned as <SELECT> node's M value. Based on a simplified document tree, the virtual layout engine derives document layout parameters without conducting actual document rendering. Final result depends on the set of sizing parameter used, placement constraints applied to each node, constraint propagation functions adopted, text layout style context employed, and the global display size setting including language encoding and user agent font families. Variation of these parameters is expected as additional aspects of document layout are considered.

[0120] Based on the display size and rendering/network capacity constraints, the document tree is partitioned into a set of sub document trees with added hyperlinks according to the layout order and content structure. Based on the sizing estimation from virtual layout, a document is partitioned and/or split according to user agent size constraints. Partitioning applies to a document and creates new documents while split operates on a document node, generating new nodes but not additional document. Partitioning and split operations are applied in accordance with the document tree to preserve the original content structure as much as possible.

[0121] Virtual layout and document partitioning are interweaved together in a bottom up process from leaf tree nodes to arrive at a set of documents where each one satisfies the user agent constraint. The steps of this process are shown in **Figure 30**. Starting with the root node, it traverses down the tree in a depth first order and accumulates document elements, calculating layout sizing information, and performing partitioning or splitting to ensure sizing constraints are satisfied for each document node collected. Sizing parameters (Wt, At, Mt, Nt) for each node T shall be partitioned or split such that 1. (Wt+Mt) < MWmax; 2. At < Amax; 3. Nt < Nmax.

[0122] Leaf node considered for sizing is either an <IMG> node **306** or a data node **308**. <IMG> node **306** cannot be split or partitioned but a scaling factor could always be found to satisfy the sizing constraint. With NoFlow flag on in the associated layout context **310**, a data node **308** cannot be split nor partitioned. Its sizing parameters are adjusted artificially **312** to satisfy the layout constraint with an assumption that the user agent would be able to make proper adjustment on the client side.

[0123] (W,N,M,A) adjustment makes updates directly on the sizing parameter values without changing the document tree. If an <IMG> node with original sizing data as (W,A,0,0) where W > MWmax or A > Amax, the sizing parameters are adjusted through a scaling factor r = min(W/MWmax, sqrt(A/Amax)). The adjusted set of sizing parameters would be (r*W, r*r*A, 0, 0). A data node under NoFlow flag with original sizing parameter (0, 0, M, N) exceeding sizing constraints would be adjusted to be (0,0, min(M, MWmax), min(N, Nmax)).

[0124] Once sizing parameters (W,A,M,N) of a node is obtained **316**, the constraint MWmax is checked and split operation **318** applied if (W+M) > MWmax until the constraint is satisfied, then both Amax and Nmax constraint are checked **320** and partition operation applied if (N > Nmax) or (A > Amax) until both are satisfied. Document partition **322** is based on node split but creating a new document tree.

[0125] To split a data node, an attempt is made to insert breaks in the longest word to bring the width requirement under the MWmax constraint. This is an update of the node without adding new ones. In the case no such break is possible, M value is artificially adjusted to MWmax with an intent for user client to handle and leave the node unchanged.

[0126] Split of non-data node T separates the original T rooted sub tree into two separate ones. This operation, denoted as split (T, N0, N1, .. Nk), requires the target node T and a set of descendant content nodes, N0, N1, ..., Nk, from its associated placement constraint. The steps are shown in **Figure 33 (a) 324**. A clone of T is created as T' **326** to be the root node of the new spin out sub tree. All paths between Ni and T are cloned in the T' rooted sub tree. Each Ni rooted sub tree is removed from the original document and inserted to T' rooted one under the same path copied **328**. A copy of placement constraint associated with T is attached to T' governing the node set N0, N1, .. Nk. A sample of split operation is shown in **Figure 33 (b) 332**, where split (T, N2, N3, N4, N5) results into two trees rooted by T **334** and T' **336** respectively. Note that clones of T, T1, T2 are created as T', T1' and T2' in this case.

[0127] A non-data node T with (W+M) > MWmax needs to be split based on columns in the associated placement constraint, as shown in **Figure 31**. It is not possible for a node with placement constraint with single column to be with (M+W) > MWmax. Every one of the descendant content node of a column will have (W+M) <= MWmax before the current node is considered. A column C (i-1) is selected **338** such that MWmax constraint is satisfied considering the partial placement constraints including all nodes belonging to columns from the first one up to C (i-1), but not when adding nodes from the next one column Ci. A new T' rooted sub tree is created **340** by node split based on nodes from maximum consecutive columns C0 up to C(i-1). In addition, a dummy <DIV> node D is also created **342** with the T rooted tree removed from the document tree. T' and T are attached to D **344** as its children with T as the next sibling of T'. D is then added back to the document tree **346** in the original place of T. Default single column placement constraint on T' and T is assigned to D **348**. A sample of this split operation is shown in **Figure 35**.

[0128] After MWmax constraint is handled, Amax and Nmax are considered as shown in **Figure 30**. If Atomic layout context flag is on **350**, such as the case for <FORM> related nodes, no document partition is allowed, and the process proceeds by updating N and A values **352** such that N <= Nmax and A <= Amax without performing any partition operation. Otherwise document is partitioned on the current node.

[0129] Steps to partition a document is shown in **Figure 32**. For a data node, a cut word is located **354** from the associated data such that number of characters before and including the cut word constitute the maximum number of consecutive words that would satisfy Nmax constraint. For a non-data node, a set of descendant content nodes is selected from rows of its placement constraint to spin off a document partition. Selecting which nodes for partition follows rows and column/row span specifications in the placement constraints. If N or A of any row is oversized **356**, a set of consecutive nodes from this row are selected to be partitioned **358**. Afterwards, it continues to examine row by row from top down **358a** and finds the maximum number of consecutive rows to spin off.

[0130] A document partition on a node is accomplished by cloning its ancestor nodes and a node split on itself, as shown in **Figure 34**. The ancestor tree path includes the closest ancestor <HTML> node **360** to its parent node. It is possible for a document node to have multiple <HTML> ancestor nodes because of expanded frame sources. If the selected <HTML> node has child <HEAD> node, the <HEAD> node rooted sub tree is also cloned **362** and attached to the new <HTML> node as descendants **364**.

[0131] Data node and non-data node are handled differently. For a data node **366**, its clone T' is created **366a** and the set of data from first characters up to the cut word identified is moved from the original node to the cloned one **366b**. An example is shown in **Figure 37**. For a non-data node **368**, a sub tree rooted by T' is created **370** by node split based on the selected descendant content nodes. T' is then attached to the cloned tree **372** path to form a document partition, rooted by an <HTML> node R'. Document changes based on tree operations according to layout sizing constraints depend on what constraints to use, how they are used and which operations to apply. Variations are possible for different considerations. For example, a new constraint Nmin and Amin could be introduced to ensure each document partition would have N and A satisfying N > Nmin or A > Amin with split and partition decisions updated accordingly. Partition could be applied accordingly with slightly different results. **Figure 36** is a document partition example.

[0132] Based on target device display size constraint, each sub document tree is scaled individually by adjusting height and width attributes through the scalar. Source image references are modified, if needed, to assure server side image transcoding capabilities, including, for example, image format change, color depth adjustment, and width/height scaling, are leveraged. Scaling process is applied to each partitioned document as well as the updated original one to change tag attributes and perform tree optimization at the same time. Scaling factor is calculated according to estimated document node layout sizing information and the target client display width available.

[0133] Overall steps for scaling are shown in **Figure 38**. Starting from the document root **374**, it assigns maximum display width available for each document node. Initially, maximum width available for the root node is assigned as the display width **376** of target client device. If scalable attributes are present **378**, scaling factor is calculated **380** and applied **382**. Scalable attributes are also applicable for text related nodes, in addition to image ones, such as WIDTH, HEIGHT, and SIZE for <INPUT> tag and COLS for <TEXTAREA> tag. SRC attribute for scaled <IMG> tags should be updated to embed scaling information with redirecting path for special image processing server when needed.

[0134] Given M, W, and Dw, scaling factor S is calculated as (Dw-M)/W if (Dw > (W+M)) and (W > 0). Otherwise, S is set to 1, i.e. the content fits the screen without the need for scaling. As M represents non-scalable sizing information such as minimum word length, only W, usually minimum image width, could be scaled.

[0135] Sizing information for a document node is updated **384** and optimized **386** after scaling operations have been performed on all descendant content child nodes according to its placement constraint. The optimization removes content nodes with empty A and N. Non-content nodes without any content offspring nodes are also deleted. Placement constraint is also simplified by removing rows and columns without any descendant content nodes. Column and row span values are updated accordingly. Whether to allow ALIGN right or left for a child <IMG> node, when present, can be determined by the available display width for the current node and the minimum width needed for the rest of child nodes. Additional constraint could be employed to eliminate document nodes that don't satisfy minimum height, width or maximum scaling factor values, for example.

[0136] Steps to assign Dw to each descendant content child node of a placement constraint are shown in **Figure 39**. The display width available for the current node, Dw **388**, its scaling factor calculated, S **388**, and a parameter to control the maximum number of iterations employed inside the steps, Maxiterate **388**, are required to proceed. For a single column placement constraint, each node of the column is assigned the same width **390** as Dw. Otherwise, an algorithm is used **392** to distribute Dw to each column, hence each node.

[0137] The objective of this algorithm is to find a set of values for all column width such that each node in the placement constraint can be accommodated and the sum of all column width equals Dw. Because of the way Dw is calculated, there always exists such a set of values. This algorithm considers first the subset of nodes with single column span. It establishes minimum column width Dm(Ci) for each column Ci. Cw, by definition, is no less than sum of these minimum widths. The difference, if there is, is distributed among each column Ci as D(Ci).

[0138] Then it iterates through all other nodes with multiple column span and makes adjustment of column width accordingly for the new node constraint while maintaining the original minimum column width assigned. Because of convergence nature of this assignment, it is expected to settle down to a solution after certain steps. However, maximum number of iteration cycles along the nodes is set **394** to arrive at an acceptable solution without much cost.

[0139] Several additional notations used in **Figure 39** warrant explanations. T is the minimum total column width based on nodes with single column span. Cn stores then number of columns. RepeatFlag indicates whether a satisfactory set of column width values have been assigned after a loop considering all nodes with multiple column span. Iterate counts the number of iterations. Ci stands for ith column and Ri for ith row. Nij is the node on ith row and jth column. With multiple column and row spans, Nij and Ni'j' could stand for the same node although i !=i' and/or j != j'. Ds represents the cumulative column width allocated defined by column span of a node. DDs is the sum of all width in addition to the minimum one for each column not covered by the column span of a node.

[0140] After a node is scaled, optimization rules are applied **386** to either remove the node or the whole rooted tree. A content document node which doesn't have any content size, i.e. A = 0 and N = 0, would be removed together with its rooted tree. In addition, <HTML> node which is not document root, created because of <FRAMESET> handling, is removed along with its child <HEAD> node rooted tree.

[0141] Content scalar as in **Figure 38** describes a method to determine how sizing specification in content elements should be updated so the resulting content would fit the display screen properly. This method considers available display window width, content sizing information estimated, and the placement constraint embedded with an algorithm to calculate the minimum width and a method to scale the applicable sizing attributes. Variations are possible for the set of content attributes to size, how new sizes are calculated, handling of boundary conditions such as when an element becomes too small to be significant, and how the algorithm is designed to arrive at a solution.

[0142] Based on the set of partitioned and scaled document trees, corresponding markup files are generated according to the subset of XHTML specs defined in Table 2, along with navigational relationship among each other. Document partition operation defines a hyperlinked relationship between the document tree with the split node and the one partitioned out. Additional ordering relationships are established for accessing one document from another in a linear manner based on the original document source text order.

[0143] Steps to calculate order for each document are shown in **Figure 40**. The order of a node T, denoted as O(T), is obtained by the sequence count of content leaf nodes appearing in the original document tree in reverse. For example, the first content leaf node is designated with order 0, the next one -1, etc. The order is propagated bottom up from leaf nodes of a document tree. The order of a node is determined by two descendant content nodes, if different, with the maximum A and maximum N sizing **396**. If these two nodes are different, the one with the larger order is selected during propagation. Many other approaches could be adopted to create hierarchical links and linear order among document trees. They could be randomized, based solely on particular type of content nodes, or using the first, last, etc. leaf node order for navigational relationship.

[0144] Sample hyperlinks and navigation order so constructed are illustrated in **Figure**

41. Six document partitions are ordered D0 **398** , D1**400**, .., D5 **402**. Hyperlink [-] points to the previous document, [+] to the next, and [^] to its parent. If the first page selected to send back to the client is based on navigation order only, the client receives document D0 **398**. The user could either click on [+] **404** from D0 **398** to go to the next page, D1**400**, or back to its hierarchical parent page, D5 **402**, through [^] **406**. From page D5 **402**, the root page, four partitions D0 **398**, D1**400**, D3 **408**, and D4 **410** are directly linked as its child document pages. Although D2 **412** follows D1**400** in order, it is also linked under D3 **408** hierarchically. Such hierarchy has been built during document partitions reflecting the original document layout semantics.

[0145] The first page returning to the client after partitioning varies depending on the need. It could be the first one based on navigation order, the root page along the partition hierarchy, or a separate page built from these partitions for special purpose. One such example is a catalog page with simple summary information on bandwidth requirement and navigation as well as hierarchy relationships among the pages, connected with hyperlinks. This will give user an overview of the target document without costing too much bandwidth resource before proceeding further.

[0146] As to a further discussion of the manner of usage and operation of the present invention, the same should be apparent from the above description. Accordingly, no further discussion relating to the manner of usage and operation will be provided.

[0147] With respect to the above description then, it is to be realized that variations and extensions of the embodiment are deemed readily apparent and obvious to one skilled in the art, and all equivalent relationships to those illustrated in the drawings and described in the specification are intended to be encompassed by the present invention.

[0148] Therefore, the foregoing is considered as illustrative only of the principles of the invention. Further, since numerous modifications and changes will readily occur to those skilled in the art, it is not desired to limit the invention to the exact construction and operation shown and described, and accordingly, all suitable modifications and equivalents may be resorted to, falling within the scope of the invention.